Jul 27, 2019

CREATING A PONG CLONE [C++]

Sometimes it's helpful to go back to the beginning. Of course we all have some dream game in our heads that we yearn to have the skills to bring to fruition, but we have to learn to walk before we can run, and recreating an old game from scratch can help us get there.

Pong is useful because it's simple (and classic), but it also has many elements that are perennial in game dev:

- Input processing
- Basic 2D rendering
- Basic 2D collision detection
- Basic 2D "physics"
- Audio

The only dependencies are SDL2, which we'll use for taking in input from the keyboard, displaying to the screen, and playing audio. SDL2 not only abstracts away the not-so-fun stuff, but it does so in a multiplatform way. The same code can run on Linux, Windows, whatever.

- **Requirements**
  - **Graphics**
  - **Control**
  - **Interactions**
  - **Scoring**
  - **Audio**
- **Setting up SDL**
- **Drawing the Net**
- **Drawing the Ball**
- **Drawing the Paddles**
- **Displaying the Scores**
- **Moving the Paddles**
- **Moving the Ball**
- **Colliding with the Paddles**
- **Colliding with the Walls**
- **Keeping Score**
- **Adding Sound Effects**

- **Conclusion**
- **Source Code**

Requirements

It helps to lay out the requirements of the game so that we can see the work cut out for us:

Graphics

- A paddle for Player One should be rendered as a white rectangle
- A paddle for Player Two should be rendered as a white rectangle
- A ball should be rendered as a white square
- A net should be rendered as a dashed line
- A score for each player should be rendered as a white number above each player's zone

Control

- WASD control Player One's paddle, which is on the left side of the screen
- Arrow keys control Player Two's paddle, which is on the right side of the screen
- The paddles should not move beyond the bounds of the screen

Interactions

- In the beginning, a ball starts in the center of the screen and then shoots off to the right
- The ball should reflect off of the top and bottom of the window
- If the ball hits the top from the left, it should reflect downward and right
- If the ball hits the top from the right, it should reflect downward and left
- If the ball hits the bottom from the left, it should reflect upward and right
- If the ball hits the bottom from the right, it should reflect upward and left
- The ball should reflect off of a player's paddle
- If the ball hits a paddle in the top third, it should reflect upward
- If the ball hits a paddle in the middle third, it

should reflect back horizontally
- If the ball hits a paddle in the botom third, its should reflect downward
- The ball should pass through the left and the right sides of the window

Scoring

- If the ball passes through the left side of the window (past Player One), it is a point for Player Two
- If the ball passes through the right side of the window (past Player Two), it is a point for Player One
- After a score occurs, the ball should start in the center and shoot off in the direction of the losing player

Audio

- If the ball hits a paddle, a sound effect should play.
- If the ball hits a wall, a sound effect should play.

> While my previous post was all about Entity Component Systems and how awesome they are, they aren't always the best choice. For a simple game like this, it would make things more complicated and less flexible to use an ECS. Instead, I'll use a more traditional approach where there is a class for a single type of entity (e.g., Paddle, Ball), all of its data is encapsulated in that class, and each object acts upon itself.
>
> "If all you have is a hammer, everything starts to look like a nail."

Setting up SDL

I'll skim over SDL2 because there are entire websites dedicated to it (**like this one**), but basically we need to create a window, create a renderer for that window, and process events that occur within the window (like keyboard input). For information about the SDL2 API, see their **documentation**.

Here's the base code:

```
```

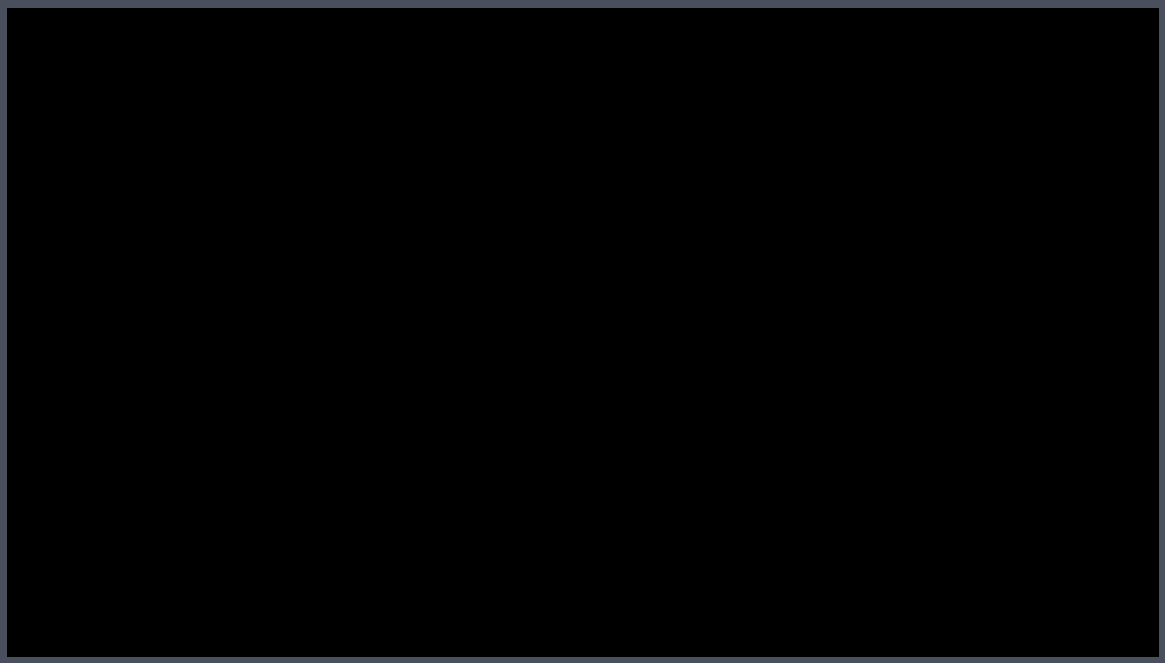
```cpp
#include <SDL2/SDL.h>

const int WINDOW_WIDTH = 1280;
const int WINDOW_HEIGHT = 720;

int main()
{
    // Initialize SDL components
    SDL_Init(SDL_INIT_VIDEO);

    SDL_Window* window = SDL_CreateWindow("Pong", 0
    SDL_Renderer* renderer = SDL_CreateRenderer(win

    // Game logic
    {
        bool running = true;

        // Continue looping and processing events u
        while (running)
        {
            SDL_Event event;
            while (SDL_PollEvent(&event))
            {
                if (event.type == SDL_QUIT)
                {
                    running = false;
                }
                else if (event.type == SDL_KEYDOWN)
                {
                    if (event.key.keysym.sym == SDL
                    {
                        running = false;
                    }
                }
            }

            // Clear the window to black
            SDL_SetRenderDrawColor(renderer, 0x0, 0
            SDL_RenderClear(renderer);

            //
            // Rendering will happen here
            //

            // Present the backbuffer
```

```
46                  SDL_RenderPresent(renderer);
47              }
48          }
49
50          // Cleanup
51          SDL_DestroyRenderer(renderer);
52          SDL_DestroyWindow(window);
53          SDL_Quit();
54
55          return 0;
56      }
```

I won't be doing any error checking to save time and space, but all of the SDL2 API calls should be checked for error codes and null pointers. See the documentation for details.

You'll notice that we clear the window and then we present the "backbuffer". Commonly in graphics, you render to a buffer that is not on screen (i.e., the backbuffer), and then when ready you swap it with the buffer currently on screen and the frontbuffer becomes the backbuffer. If you rendered directly into the buffer that the player was looking at, they would see things being drawn and it would break the illusion.

Here is the result:



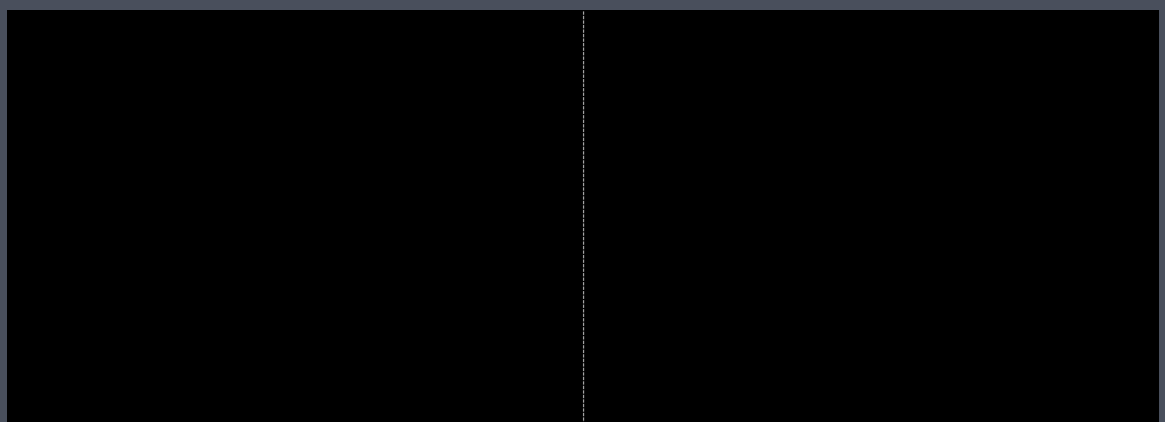
Drawing the Net

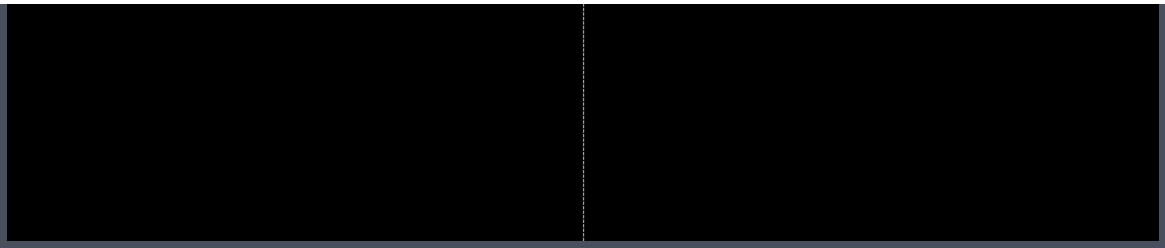
Let's start with the simplest possible thing we can get on screen: the net. It has no logic to it; it's just a dashed line running from the top of the screen to the bottom.

Insert the code between the clearing and presenting.

```
    // Clear the window to black
    SDL_SetRenderDrawColor(renderer, 0x0, 0x0, 0x0, 0xF
    SDL_RenderClear(renderer);

    // Set the draw color to be white
    SDL_SetRenderDrawColor(renderer, 0xFF, 0xFF, 0xFF,

    // Draw the net
    for (int y = 0; y < WINDOW_HEIGHT; ++y)
    {
        if (y % 5)
        {
            SDL_RenderDrawPoint(renderer, WINDOW_WIDTH
        }
    }

    // Present the backbuffer
    SDL_RenderPresent(renderer);
```

It loops through the height of the window and draws the net at **(WINDOW_WIDTH/2, y)** whenever **y** is not a multiple of five. That leaves a gap every 5 pixels, creating the dashed line. Note that the origin is in the upper left corner of the screen, so that **x** gets bigger towards the right and **y** gets bigger towards the bottom. This will be important later.





Drawing the Ball

The ball is the next simplest thing to get on the screen, but it’s more involved because we’re going to create a class for it.

But first, let’s create a class to encapsulate the idea of a 2-dimensionsal vector: **Vec2**. It has two floats, **x** and **y**, which we can use to represent position and (later) velocity.

```
class Vec2
{
public:
    Vec2()
        : x(0.0f), y(0.0f)
    {}

    Vec2(float x, float y)
        : x(x), y(y)
    {}

    Vec2 operator+(Vec2 const& rhs)
    {
        return Vec2(x + rhs.x, y + rhs.y);
    }

    Vec2& operator+=(Vec2 const& rhs)
    {
        x += rhs.x;
        y += rhs.y;

        return *this;
    }

    Vec2 operator*(float rhs)
    {
        return Vec2(x * rhs, y * rhs);
    }

    float x, y;
};
```

Some convenient operators are overloaded so that we can do something like **position += velocity * time**, and it knows how to apply addition and multiplication to both **x** and **y**.

Now that we have **Vec2**, we can create **Ball**:

```
const int BALL_WIDTH = 15;
const int BALL_HEIGHT = 15;

class Ball
{
public:
    Ball(Vec2 position)
        : position(position)
    {
        rect.x = static_cast<int>(position.x);
        rect.y = static_cast<int>(position.y);
        rect.w = BALL_WIDTH;
        rect.h = BALL_HEIGHT;
    }

    void Draw(SDL_Renderer* renderer)
    {
        rect.x = static_cast<int>(position.x);
        rect.y = static_cast<int>(position.y);

        SDL_RenderFillRect(renderer, &rect);
    }

    Vec2 position;
    SDL_Rect rect{};
};
```

I don't use getters or setters anywhere in this code because I think they should be reserved for when some sort of logic is required (e.g., incrementing a pointer before returning a value), rather than a simple **x = val** or **return x**.

They also make math operations hard to read.

Simple rectangular geometry is describe in SDL2 as an **SDL_Rect**, which describes an object's width and height, and its position in 2D space. We initialize the ball with a position and it fills out the **SDL_Rect**. When it goes to draw itself, it updates the **SDL_Rect**'s position with the ball object's position, and tells SDL to render it to the backbuffer.

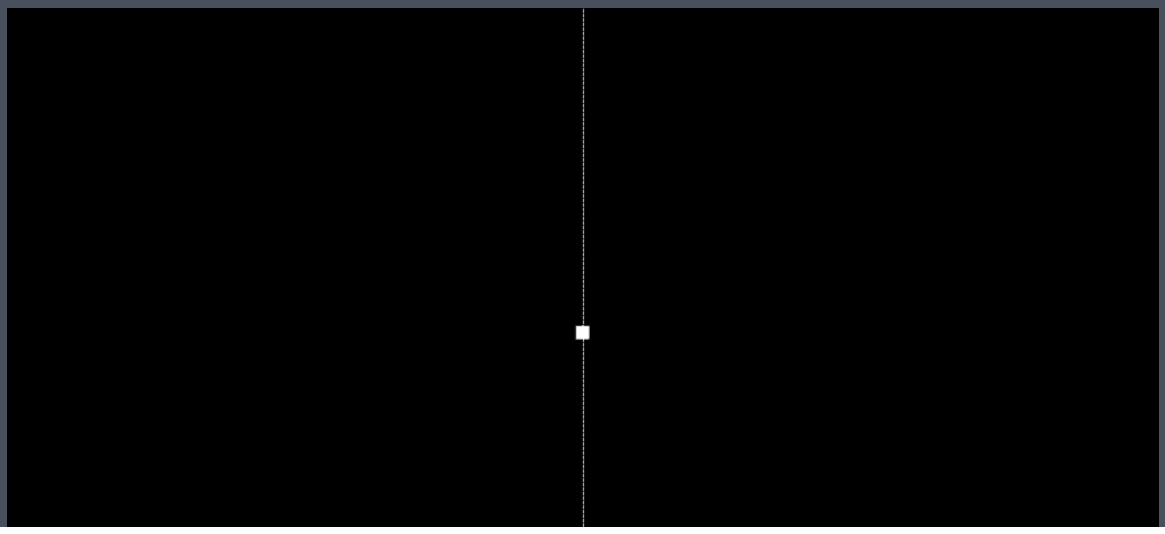
Ideally we wouldn't want something like a **Ball** to need to

know about something like **SDL_Renderer**. In a bigger game we'd like to keep **Paddle**s and **Ball**s ignorant of the rendering system, but for Pong it would add unnecessary complexity.

We then need to create a ball object and draw it:

```
    SDL_Init(SDL_INIT_VIDEO);

    SDL_Window* window = SDL_CreateWindow("Pong", 0, 0,
    SDL_Renderer* renderer = SDL_CreateRenderer(window,

    // Create the ball
    Ball ball(
        Vec2((WINDOW_WIDTH / 2.0f) - (BALL_WIDTH / 2.0f
        (WINDOW_HEIGHT / 2.0f) - (BALL_WIDTH / 2.0f)));

    [...]

    // Draw the ball
    ball.Draw(renderer);

    // Present the backbuffer
    SDL_RenderPresent(renderer);
```

The ball's initial position is set to the middle of the screen. Half of the ball's width and height are subtracted from the **x** and **y** values because SDL2 considers the origin of an object to be the upper left corner. If we don't do the subtraction then they actually start off center which is fine, but aesthetically unappealing.



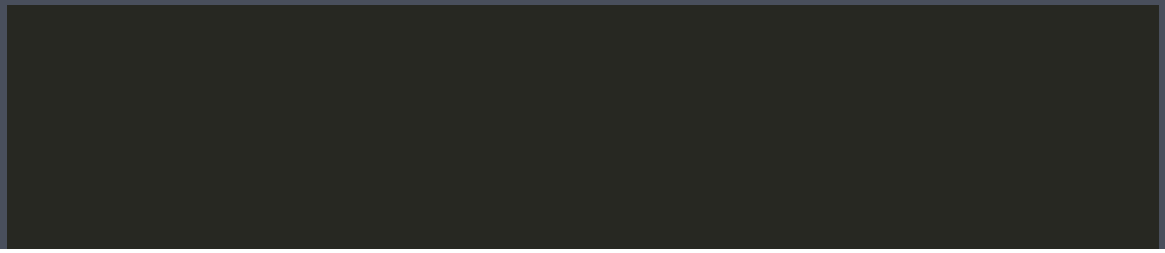
Drawing the Paddles

Now for the paddles. We'll create a class for these just like we did for the ball. It looks very similar to the **Ball** class for now, but that will change later.

```
class Paddle
{
public:
    Paddle(Vec2 position)
        : position(position)
    {
        rect.x = static_cast<int>(position.x);
        rect.y = static_cast<int>(position.y);
        rect.w = PADDLE_WIDTH;
        rect.h = PADDLE_HEIGHT;
    }

    void Draw(SDL_Renderer* renderer)
    {
        rect.y = static_cast<int>(position.y);

        SDL_RenderFillRect(renderer, &rect);
    }

    Vec2 position;
    SDL_Rect rect{};
};
```

We only bother changing **rect.y** when drawing because we don't ever want the paddles to move right or left on the x axis.

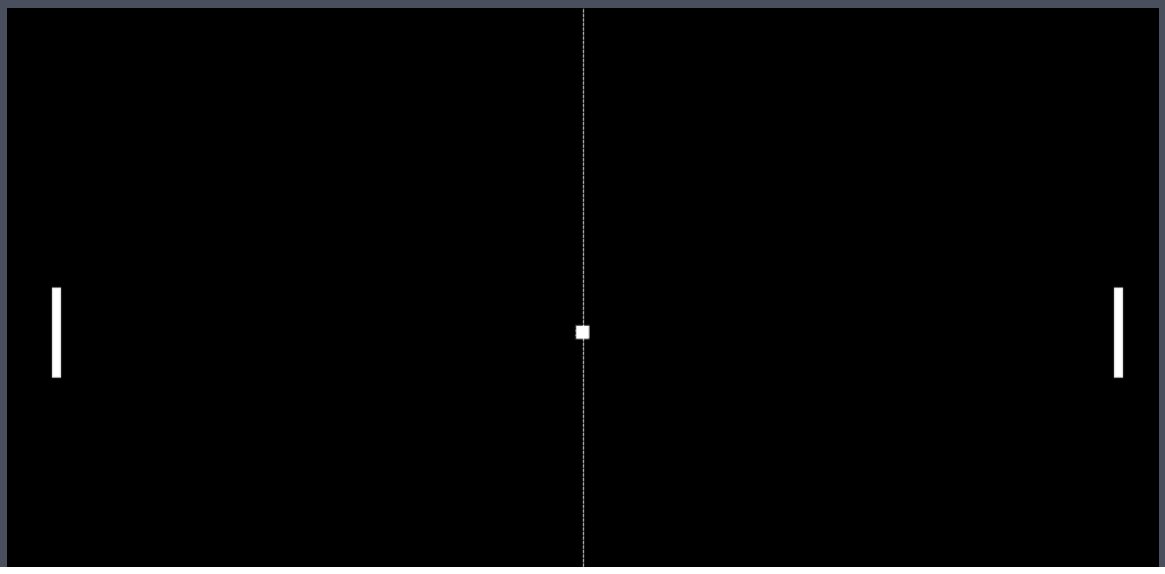
Then of course we create both paddles and draw them.



```
    // Create the ball
    Ball ball(
        Vec2(WINDOW_WIDTH / 2.0f, WINDOW_HEIGHT / 2.0f)

    // Create the paddles
    Paddle paddleOne(
        Vec2(50.0f, (WINDOW_HEIGHT / 2.0f) - (PADDLE_HE

    Paddle paddleTwo(
        Vec2(WINDOW_WIDTH - 50.0f, (WINDOW_HEIGHT / 2.0

    [...]

    // Draw the ball
    ball.Draw(renderer);

    // Draw the paddles
    paddleOne.Draw(renderer);
    paddleTwo.Draw(renderer);

    // Present the backbuffer
    SDL_RenderPresent(renderer);
```

paddleOne is created with an initial position of **x=50.0** so that it's slightly away from the screen edge. Like the ball, we set its initial **y** to the middle, and compensate for the corner origin by subtracting its height.

paddleTwo is the same on the y axis, but its **x** is instead 50 units away from the right side of the screen.



Displaying the Scores

The last thing we need to put on the screen before diving into gameplay logic is the player scores. It will be another class, and it will interact with the text rendering piece of SDL2: **SDL2_ttf**.

We first need to supply a font for SDL2 to use. I'm using **Deja Vu Sans Mono** because it's free. You can get it **here**. Unzip and place it in the same directory as your compiled binary.

There is some SDL2 setup that needs to happen for the font to be usable:

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>

[...]

int main()
{
    // Initialize SDL components
    SDL_Init(SDL_INIT_VIDEO);
    TTF_Init();

    SDL_Window* window = SDL_CreateWindow("Pong", 0
    SDL_Renderer* renderer = SDL_CreateRenderer(win

    // Initialize the font
    TTF_Font* scoreFont = TTF_OpenFont("DejaVuSansM

    [...]

    // Cleanup
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    TTF_CloseFont(scoreFont)
    TTF_Quit();
    SDL_Quit();

    return 0;
}
```

We now need the **PlayerScore** class which is responsible for managing the resources required to display text.

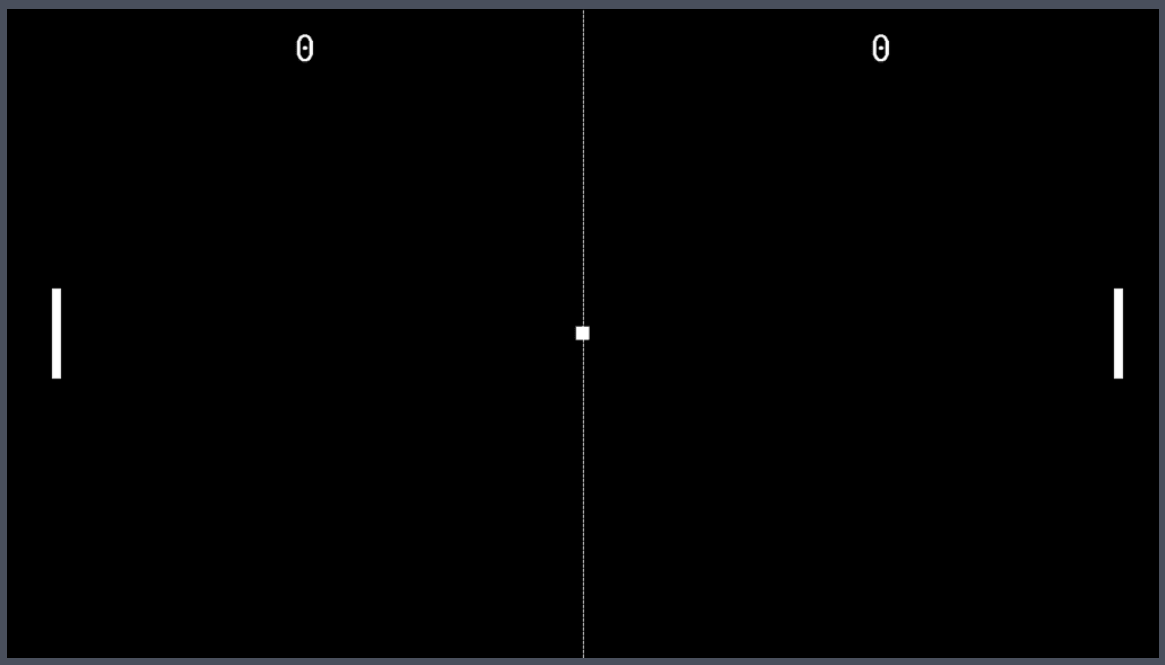
```
class PlayerScore
{
public:
    PlayerScore(Vec2 position, SDL_Renderer* render
        : renderer(renderer), font(font)
    {
        surface = TTF_RenderText_Solid(font, "0", {
        texture = SDL_CreateTextureFromSurface(rend

        int width, height;
        SDL_QueryTexture(texture, nullptr, nullptr,

        rect.x = static_cast<int>(position.x);
        rect.y = static_cast<int>(position.y);
        rect.w = width;
        rect.h = height;
    }

    ~PlayerScore()
    {
        SDL_FreeSurface(surface);
        SDL_DestroyTexture(texture);
    }

    void Draw()
    {
        SDL_RenderCopy(renderer, texture, nullptr,
    }

    SDL_Renderer* renderer;
    TTF_Font* font;
    SDL_Surface* surface{};
    SDL_Texture* texture{};
    SDL_Rect rect{};
};
```

Essentially, SDL2 writes the (white) text onto an **SDL_Surface** and then creates a texture from that surface which is what is actually drawn to the screen. Any time the score changes, a new surface and texture must be created, and the old ones are destroyed. The class keeps a pointer to **renderer** and **font** because they're needed whenever the score is updated.

We can now display the scores.

```
    // Initialize the font
    TTF_Font* scoreFont = TTF_OpenFont("DejaVuSansMono.

    // Create the player score text fields
    PlayerScore playerOneScoreText(Vec2(WINDOW_WIDTH /

    PlayerScore playerTwoScoreText(Vec2(3 * WINDOW_WIDT

    [...]

    // Draw the paddles
    paddleOne.Draw(renderer);
    paddleTwo.Draw(renderer);

    // Display the scores
    playerOneScoreText.Draw();
    playerTwoScoreText.Draw();
```

playerOneScoreText is set to a position 20 units down from the top of the screen and 1/4 of the way across, while **playerTwoScoreText** is set to a position 20 units down from the top of the screen and 3/4 of the way across.



Moving the Paddles

Finally it's time to make something move. We need to take keyboard input and translate it into paddle movement. The

paddles can only move along the y-axis, and they should not move beyond the bounds of the screen.

First, let's add a new field to the **Paddle** class that stores the paddle's current velocity. Velocity for our purposes simply indicates three states: moving up, moving down, or not moving at all. If moving, it happens at a constant speed (i.e., no acceleration).

We also need to add an **Update** method to **Paddle** that updates its position as a function of its velocity and the time that passed between the last frame and the current frame: **dt**. If you don't use **dt**, then changes in framerate will affect the speed of the paddle's movement. By multiplying by **dt**, you always get **PADDLE_SPEED** units of movement per second.

Paddle now looks like this:

```
class Paddle
{
public:
    Paddle(Vec2 position, Vec2 velocity)
        : position(position), velocity(velocity)
    {
        rect.x = static_cast<int>(position.x);
        rect.y = static_cast<int>(position.y);
        rect.w = PADDLE_WIDTH;
        rect.h = PADDLE_HEIGHT;
    }

    void Update(float dt)
    {
        position += velocity * dt;

        if (position.y < 0)
        {
            // Restrict to top of the screen
            position.y = 0;
        }
        else if (position.y > (WINDOW_HEIGHT - PADD
        {
            // Restrict to bottom of the screen
            position.y = WINDOW_HEIGHT - PADDLE_HEI
        }
    }

    void Draw(SDL_Renderer* renderer)
    {
        rect.y = static_cast<int>(position.y);

        SDL_RenderFillRect(renderer, &rect);
    }

    Vec2 position;
    Vec2 velocity;
    SDL_Rect rect{};
};
```

When checking to see if the paddle is moving off the bottom of the screen, we again compensate for the origin being in the upper left corner.

We also need to now track the time between frames, which is quick and easy.

```
#include <chrono>
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>

[...]

bool running = true;

float dt = 0.0f;

while (running)
{
    auto startTime = std::chrono::high_resolution_c

    [...]

    // Calculate frame time
    auto stopTime = std::chrono::high_resolution_cl
    dt = std::chrono::duration<float, std::chrono::
}
```

Before we can add the code to get the input, we need to modify our paddle instantiations to pass in the initial paddle velocities. We want them to zero to start.

```
[...]

// Create the paddles
Paddle paddleOne(
    Vec2(50.0f, WINDOW_HEIGHT / 2.0f),
    Vec2(0.0f, 0.0f));

Paddle paddleTwo(
    Vec2(WINDOW_WIDTH - 50.0f, WINDOW_HEIGHT / 2.0f
    Vec2(0.0f, 0.0f));

[...]
```

Now we check for input on WASD and the arrow keys. We already received keyboard input when we used the Escape key

to quit the game, and getting the other keys is no different. However, we also want to know when a key is released, and we need to store each key's state so we can check it later. We'll use an array of bools where each index corresponds to the four inputs (enumerated in **Buttons**): paddle one moving up, paddle one moving down, paddle two moving up, and paddle two moving down.

After checking for input, we then tell each paddle to update their positions.

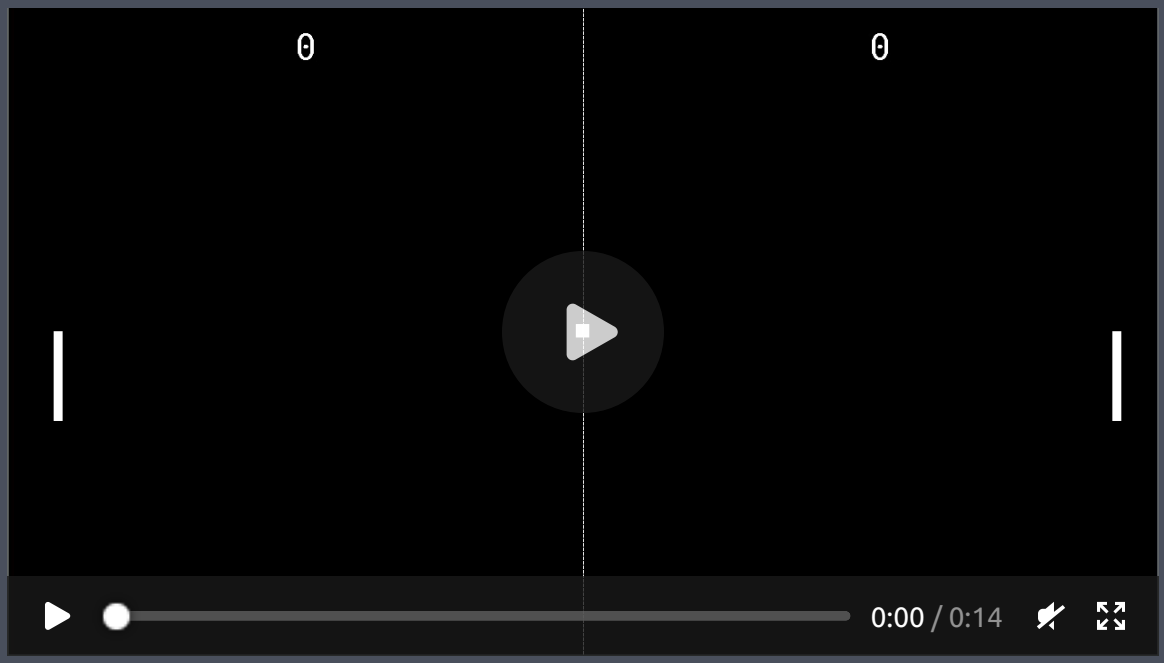
```
enum Buttons
{
    PaddleOneUp = 0,
    PaddleOneDown,
    PaddleTwoUp,
    PaddleTwoDown,
};

const float PADDLE_SPEED = 1.0f;

[...]

bool running = true;
bool buttons[4] = {};

float dt= 0.0f;

while (running)
{
    auto startTime = std::chrono::high_resolution_

    SDL_Event event;
    while (SDL_PollEvent(&event))
    {
        if (event.type == SDL_QUIT)
        {
            running = false;
        }
        else if (event.type == SDL_KEYDOWN)
        {
            if (event.key.keysym.sym == SDLK_ESCAP
            {
                running = false;
            }
            else if (event.key.keysym.sym == SDLK_
            {
                buttons[Buttons::PaddleOneUp] = tr
            }
            else if (event.key.keysym.sym == SDLK_
            {
                buttons[Buttons::PaddleOneDown] =
            }
            else if (event.key.keysym.sym == SDLK_
            {
                buttons[Buttons::PaddleTwoUp] = tr
```

```
            }
            else if (event.key.keysym.sym == SDLK_
            {
                buttons[Buttons::PaddleTwoDown] =
            }
        }
        else if (event.type == SDL_KEYUP)
        {
            if (event.key.keysym.sym == SDLK_w)
            {
                buttons[Buttons::PaddleOneUp] = fa
            }
            else if (event.key.keysym.sym == SDLK_
            {
                buttons[Buttons::PaddleOneDown] =
            }
            else if (event.key.keysym.sym == SDLK_
            {
                buttons[Buttons::PaddleTwoUp] = fa
            }
            else if (event.key.keysym.sym == SDLK_
            {
                buttons[Buttons::PaddleTwoDown] =
            }
        }
    }

    if (buttons[Buttons::PaddleOneUp])
    {
        paddleOne.velocity.y = -PADDLE_SPEED;
    }
    else if (buttons[Buttons::PaddleOneDown])
    {
        paddleOne.velocity.y = PADDLE_SPEED;
    }
    else
    {
        paddleOne.velocity.y = 0.0f;
    }

    if (buttons[Buttons::PaddleTwoUp])
    {
        paddleTwo.velocity.y = -PADDLE_SPEED;
    }
    else if (buttons[Buttons::PaddleTwoDown])
    {
        paddleTwo.velocity.y = PADDLE_SPEED;
```

```
92              paddleTwo.velocity.y = PADDLE_SPEED;
93          }
94          else
95          {
96              paddleTwo.velocity.y = 0.0f;
97          }
98
99          // Update the paddle positions
100         paddleOne.Update(dt);
101         paddleTwo.Update(dt);
102
103         [...]
104     }
```

The **Buttons** enum is not an **enum class** (the more modern way of using an enum for better type-safety) because it is used as an index into an array which isn't allowed with an **enum class**.

And now we have paddle movement!



Moving the Ball

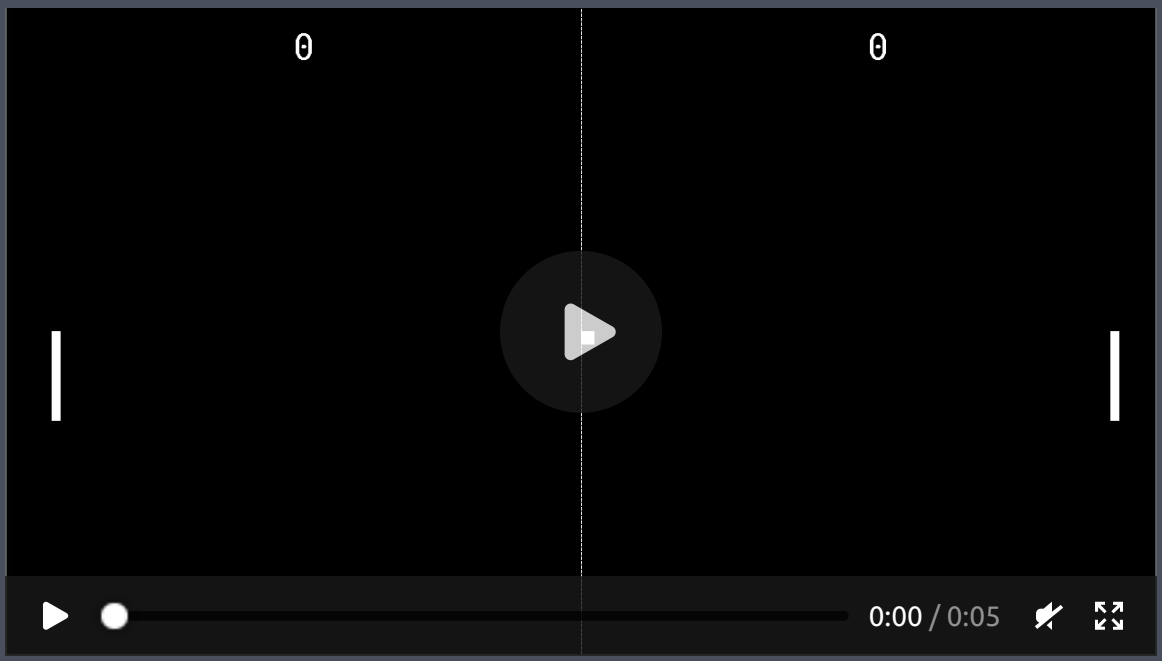
To get the ball moving, we need need to add a velocity component to it like we did for the paddles.

```
class Ball
{
public:
    Ball(Vec2 position, Vec2 velocity)
        : position(position), velocity(velocity)
    {
        rect.x = static_cast<int>(position.x);
        rect.y = static_cast<int>(position.y);
        rect.w = BALL_WIDTH;
        rect.h = BALL_HEIGHT;
    }

    void Update(float dt)
    {
        position += velocity * dt;
    }

    void Draw(SDL_Renderer* renderer)
    {
        rect.x = static_cast<int>(position.x);
        rect.y = static_cast<int>(position.y);

        SDL_RenderFillRect(renderer, &rect);
    }

    Vec2 position;
    Vec2 velocity;
    SDL_Rect rect{};
};
```

The velocity of the ball won't be controlled by the players of course, but instead will change depending on the ball's interactions with the paddles and the walls. To start we'll just add some initial velocity.

```
1     [...]
2
3     const float BALL_SPEED = 1.0f;
4
5     [...]
6
7     // Create the ball
8     Ball ball(
9         Vec2(WINDOW_WIDTH / 2.0f, WINDOW_HEIGHT / 2.0f)
10        Vec2(BALL_SPEED, 0.0f));
11
12    [...]
13
14    // Update the paddle positions
15    paddleOne.Update(dt);
16    paddleTwo.Update(dt);
17
18    // Update the ball position
19    ball.Update(dt);
20
21    [...]
```

The game will start with the ball heading to the right.



Colliding with the Paddles

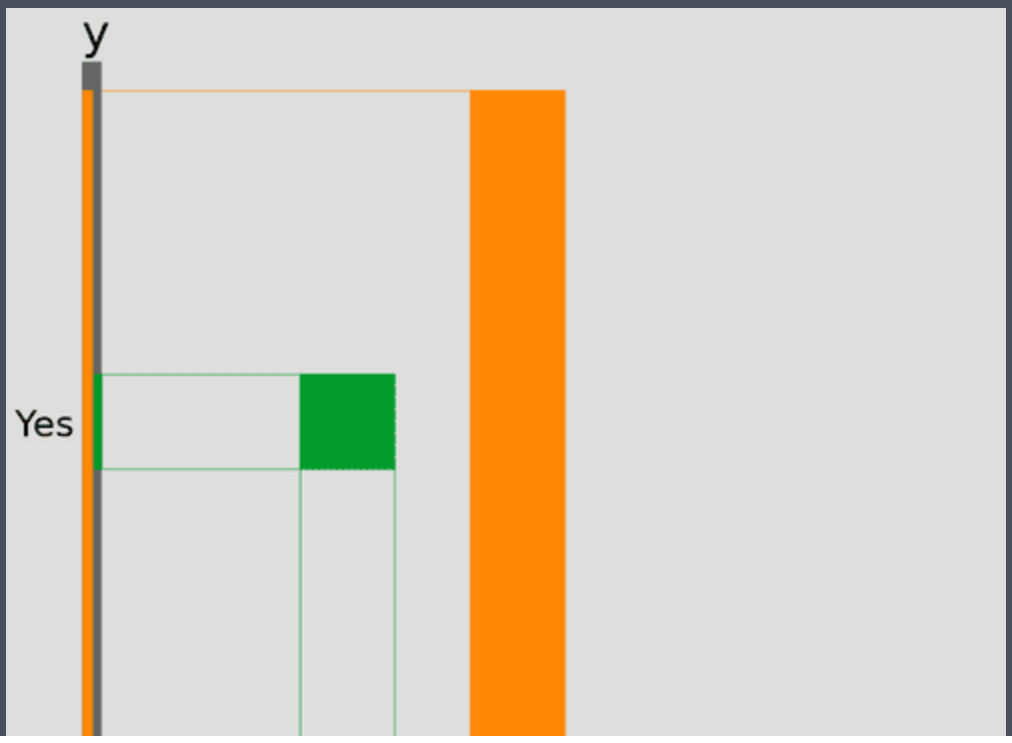
Now the fun stuff. We have everything drawing on screen, we now only need to have the ball interact with the world to get the actual gameplay going.

Collision detection is a vast and complicated topic. Fortunately for us it's rather simple for Pong because of two things: it's 2D (3D gets much more complicated), and everything in the game is aligned to the axes (the sides of the paddles and the sides of the ball are all parallel with either the x- or the y-axis).

To detect a collision between the balls and the paddles, we'll make use of something called the **Separating Axis Theorem (SAT)**. The SAT says (in simplified terms) that if you can show that the projections of two objects onto an axis have a gap, then the objects are not colliding. It gets pretty complicated if your 2D objects are rotated or you're working in 3D, but for us it's simpler.

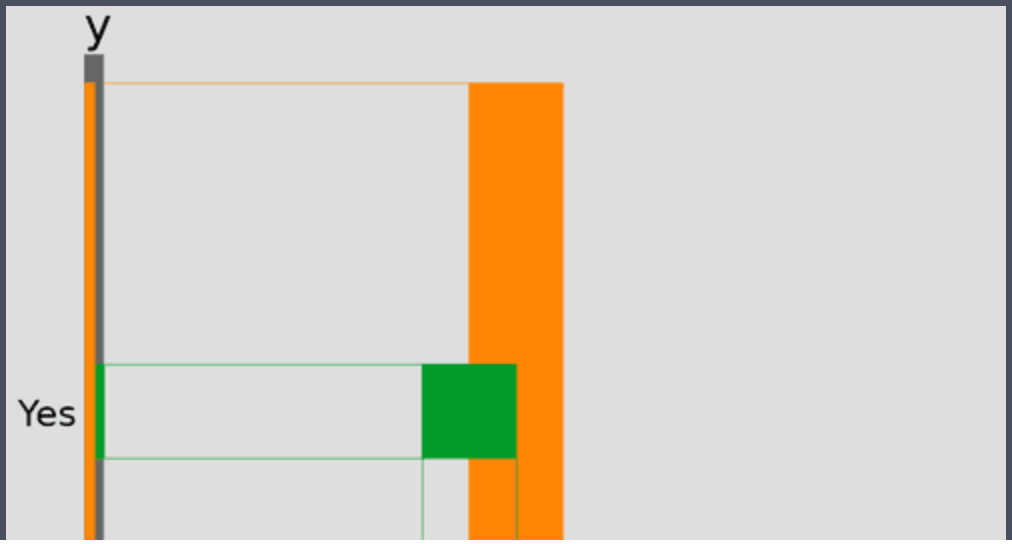
What we want to do is project (cast a shadow) from each object to each axis, so project from the ball onto the x- and y-axis, and project from the paddle onto the x- and y-axis. If we can find even a single axis on which the projections don't touch, then the objects are not colliding.

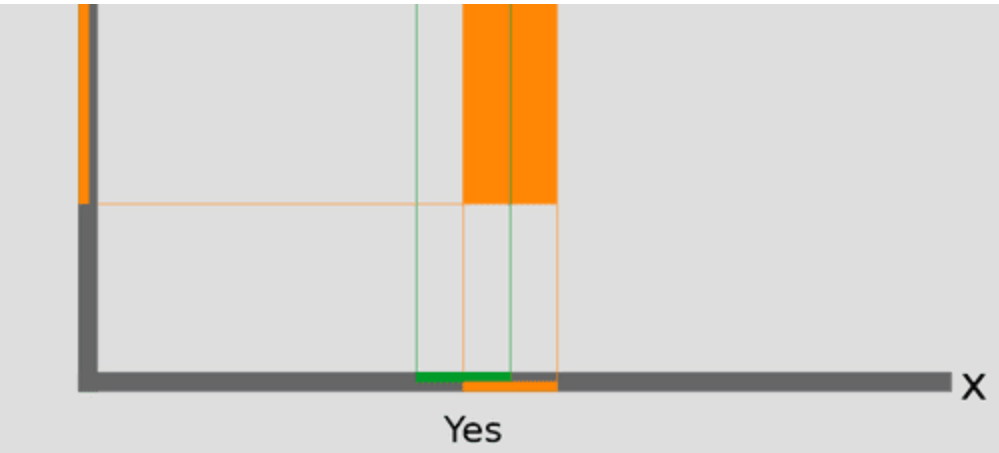
In the following diagrams, the lines connect the objects to their projections. When orange and green overlap, there is a *possible* collision. Only if both cases overlap is there actually a collision.



x
No

y
No
x
Yes


y
Yes

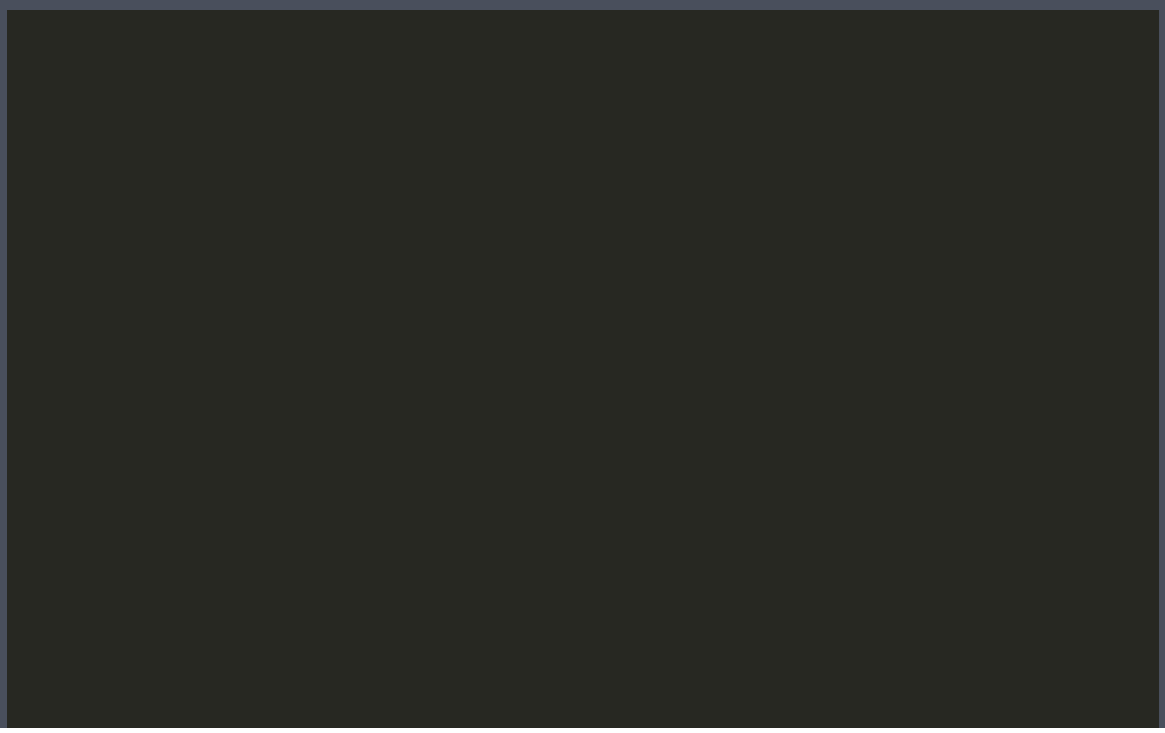


In the first two examples, there are overlapping projections on a single axis but not both. In the third example there is overlapping on both and the two objects are in fact colliding.

In code, to “project” onto the axes, we actually just need x-coordinates of the left and right of the paddle and ball, and the y-coordinates of the top and bottom of the paddle and ball. The top and the bottom are projected onto the y-axis, and the left and the right are projected onto the x-axis (look at the examples above again).

Remember that the origins of each are in the top left corner. The math would be different otherwise.

We then check for each of the four possible gaps that could occur, and if any of them exist, there is no collision.



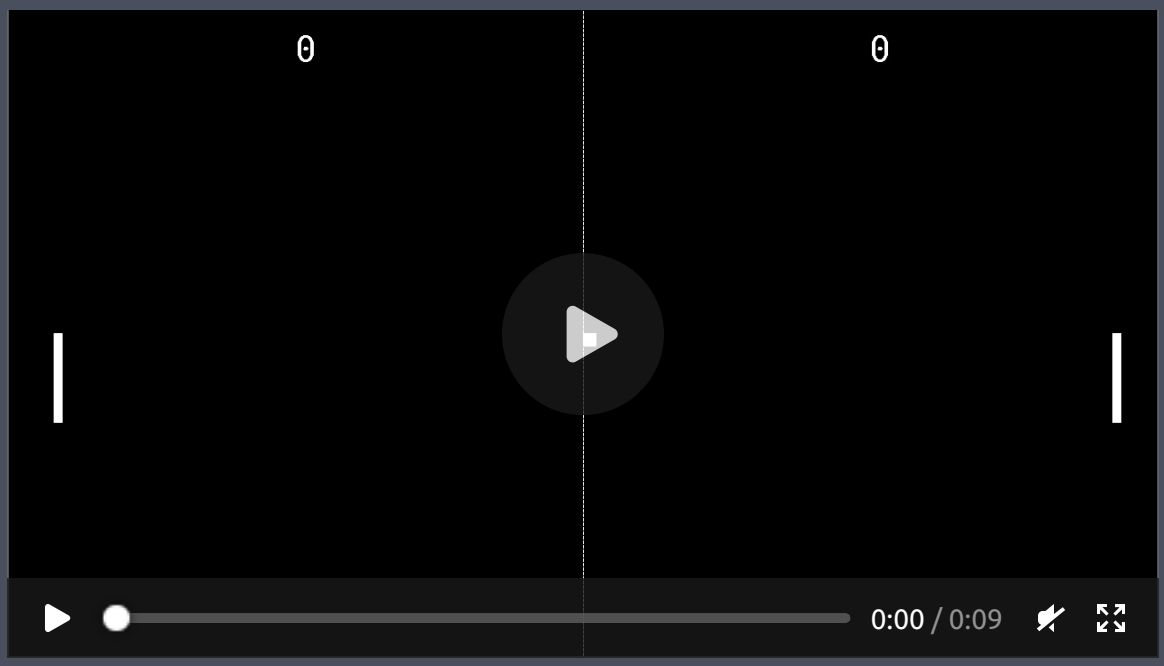
```
bool CheckPaddleCollision(Ball const& ball, Paddle
{
    float ballLeft = ball.position.x;
    float ballRight = ball.position.x + BALL_WIDTH;
    float ballTop = ball.position.y;
    float ballBottom = ball.position.y + BALL_HEIGH

    float paddleLeft = paddle.position.x;
    float paddleRight = paddle.position.x + PADDLE_
    float paddleTop = paddle.position.y;
    float paddleBottom = paddle.position.y + PADDLE

    if (ballLeft >= paddleRight)
    {
        return false;
    }

    if (ballRight <= paddleLeft)
    {
        return false;
    }

    if (ballTop >= paddleBottom)
    {
        return false;
    }

    if (ballBottom <= paddleTop)
    {
        return false;
    }

    return true;
}
```

In the loop, We check for the ball colliding with each paddle and, if there's a hit, send the ball back the other direction by negating its velocity along the x-axis. A simple reflection.

```
    [...]

    // Update the paddle positions
    paddleOne.Update(dt);
    paddleTwo.Update(dt);

    // Update the ball position
    ball.Update(dt);

    // Check collisions
    if (CheckPaddleCollision(ball, paddleOne) ||
        CheckPaddleCollision(ball, paddleTwo))
    {
        ball.velocity.x = -ball.velocity.x;
    }

    [...]
```

The reflections are so easy to calculate because everything is axially aligned. When rotations enter the mix, you have to bring in linear algebra.

Which gives us this:



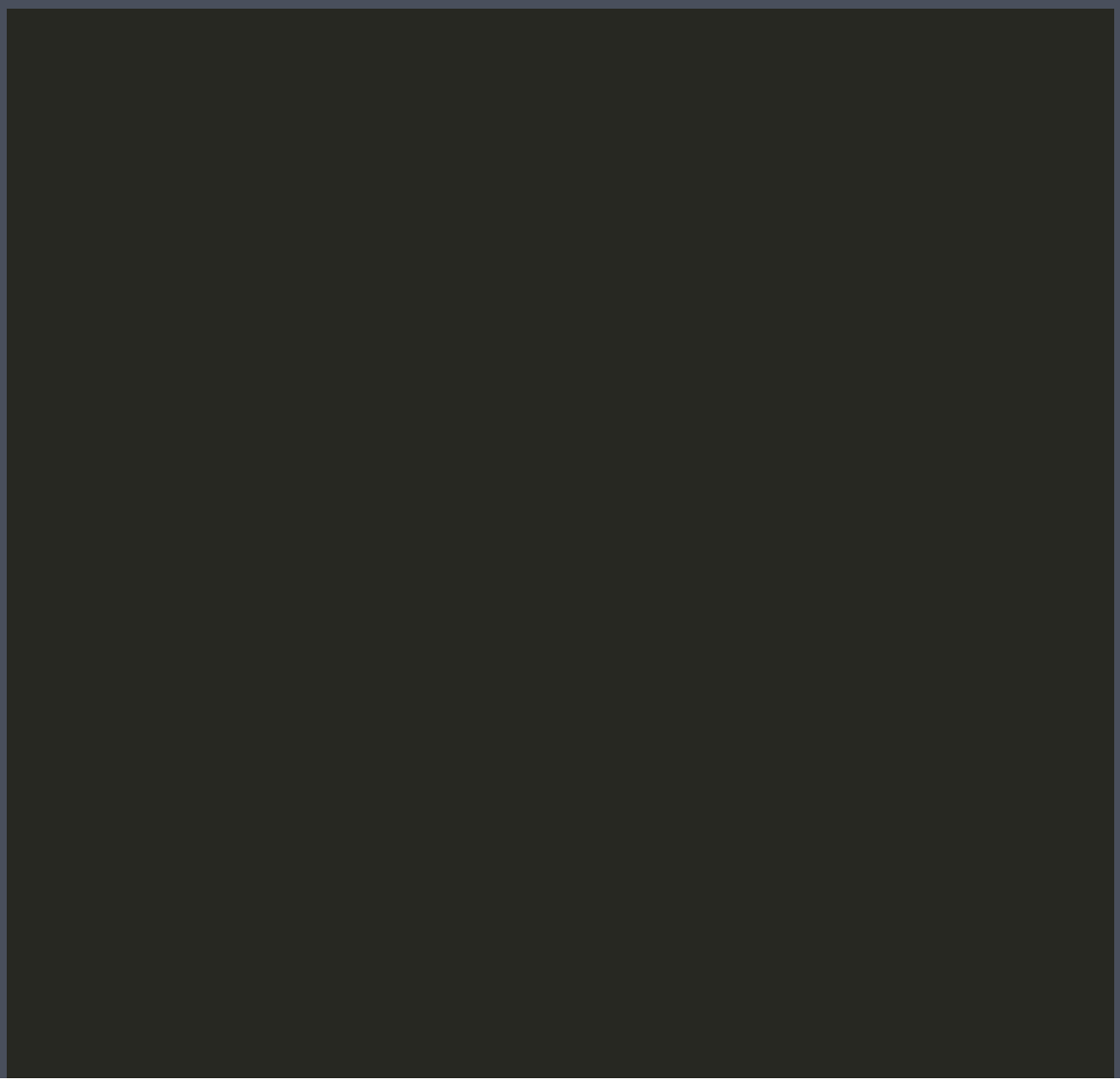
Well that's pretty boring. If you look at a video of the old Pong, the ball bounces at an extreme angle when it hits

a paddle, and the direction is dependent on where on the paddle it hits. So let's do that. If it hits the paddle in the middle, it'll reflect back like in the video, but if it hits at the top it'll reflect upward and if it hits at the bottom it'll reflect downward.

We also need to move the ball outside of the paddle when it penetrates. If we don't do that, the ball will sometimes get stuck in the paddle when a collision is detected and the ball's velocity is reversed, but then another collision is detected before it leaves the paddle, resulting in a weird oscillatory effect.

We'll create an enum to enumerate the types of collisions, and a struct to hold that type as well as the penetration amount, and then return that from the collision-checking function instead.

We'll also add a **CollideWithPaddle** method to **Ball** to keep **Ball** operating on its own data.



```
enum class CollisionType
{
    None,
    Top,
    Middle,
    Bottom
};

struct Contact
{
    CollisionType type;
    float penetration;
};

[...]

class Ball
{
    [...]

    void CollideWithPaddle(Contact const& contact)
    {
        position.x += contact.penetration;
        velocity.x = -velocity.x;

        if (contact.type == CollisionType::Top)
        {
            velocity.y = -.75f * BALL_SPEED;
        }
        else if (contact.type == CollisionType::Bo
        {
            velocity.y = 0.75f * BALL_SPEED;
        }
    }

    [...]
};

[...]

Contact CheckPaddleCollision(Ball const& ball, Pad
{
    float ballLeft = ball.position.x;
    float ballRight = ball.position.x + BALL_WIDTH
    float ballTop = ball.position.y;
```

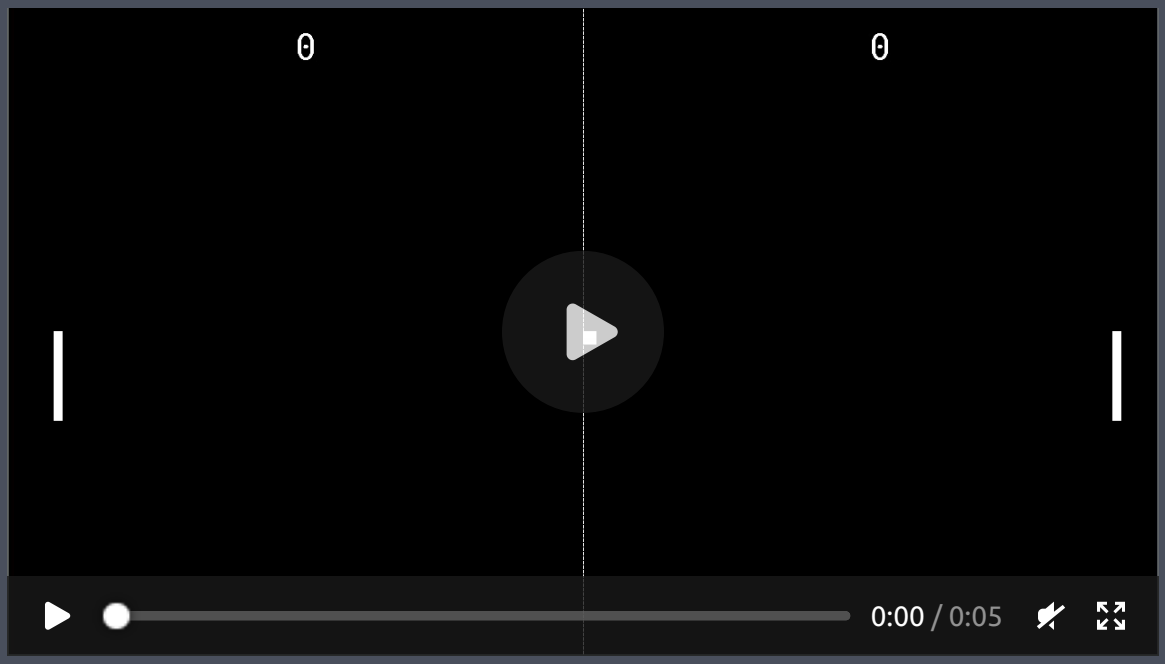
```
46          float ballBottom = ball.position.y + BALL_HEIG
47
48          float paddleLeft = paddle.position.x;
49          float paddleRight = paddle.position.x + PADDLE
50          float paddleTop = paddle.position.y;
51          float paddleBottom = paddle.position.y + PADDL
52
53          Contact contact{};
54
55          if (ballLeft >= paddleRight)
56          {
57              return contact;
58          }
59
60          if (ballRight <= paddleLeft)
61          {
62              return contact;
63          }
64
65          if (ballTop >= paddleBottom)
66          {
67              return contact;
68          }
69
70          if (ballBottom <= paddleTop)
71          {
72              return contact;
73          }
74
75          float paddleRangeUpper = paddleBottom - (2.0f
76          float paddleRangeMiddle = paddleBottom - (PADD
77
78          if (ball.velocity.x < 0)
79          {
80              // Left paddle
81              contact.penetration = paddleRight - ballLe
82          }
83          else if (ball.velocity.x > 0)
84          {
85              // Right paddle
86              contact.penetration = paddleLeft - ballRig
87          }
88
89          if ((ballBottom > paddleTop)
90              && (ballBottom < paddleRangeUpper))
91          {
92              contact.type = CollisionType::Top;
```

```
92              contact.type = CollisionType::Top;
93          }
94          else if ((ballBottom > paddleRangeUpper)
95               && (ballBottom < paddleRangeMiddle))
96          {
97              contact.type = CollisionType::Middle;
98          }
99          else
100         {
101             contact.type = CollisionType::Bottom;
102         }
103
104         return contact;
105     }
106
107     [...]
108
109     // Update the paddle positions
110     paddleOne.Update(dt);
111     paddleTwo.Update(dt);
112
113     // Update the ball position
114     ball.Update(dt);
115
116     // Check collisions
117     if (Contact contact = CheckPaddleCollision(ball, p
118         contact.type != CollisionType::None)
119     {
120         ball.CollideWithPaddle(contact);
121     }
122     else if (contact = CheckPaddleCollision(ball, padd
123         contact.type != CollisionType::None)
124     {
125         ball.CollideWithPaddle(contact);
126     }
```

Why a **struct** and not a **class**? Even though all of the classes so far have been entirely public (which is the same as a **struct**), I like to reserve the term **struct** for Plain Old Data with no logic of any kind, while a **class** is a bundle of logic and data together.

The first item in **CollisionType** is **None**, so by initializing a **Contact** to default values with **{}**, its **type** is implicitly set to **None**, which is why we're able to return it immediately when we find a gap via the **SAT**.

We reflect differently depending on the paddle zone (Top, Bottom, or Middle). Multiplying the ball's speed by 0.75 gives it a nice sharp angle (assuming a speed of 1.0, then the resultant vector is [1.0, 0.75]).

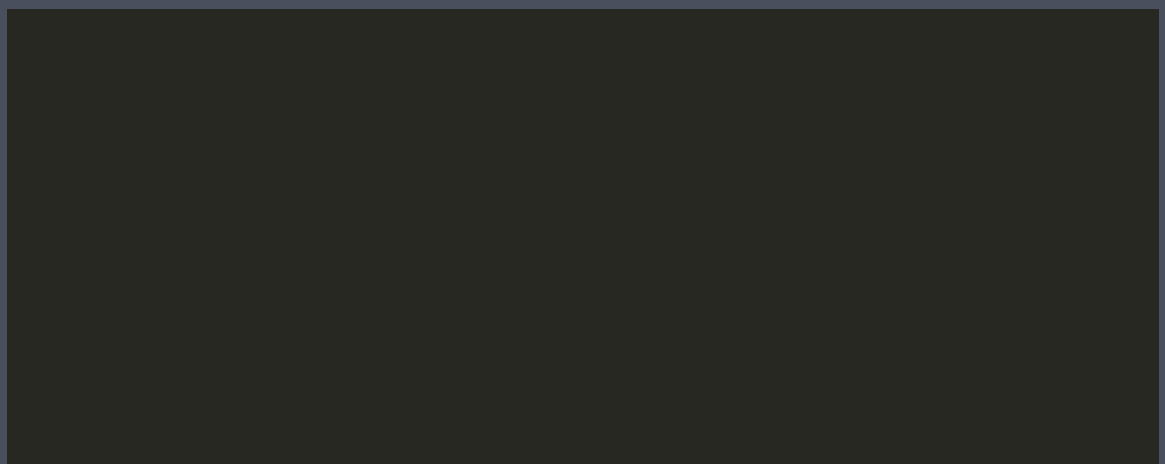


Much better, but we now we need wall collision.

Colliding with the Walls

Colliding with the wall is a lot simpler than colliding with a paddle because the extents and position of the walls are always the same: **(0,0)** to **(WINDOW_WIDTH,WINDOW_HEIGHT)**. We just check the edges of the ball with the left, right, top, and bottom of the window.

We can use the same **CollisionType** enum as before, we just need to add **Left** and **Right** to it.



```
enum class CollisionType
{
    None,
    Top,
    Middle,
    Bottom,
    Left,
    Right
};

[...]

Contact CheckWallCollision(Ball const& ball)
{
    float ballLeft = ball.position.x;
    float ballRight = ball.position.x + BALL_WIDTH;
    float ballTop = ball.position.y;
    float ballBottom = ball.position.y + BALL_HEIGH

    Contact contact{};

    if (ballLeft < 0.0f)
    {
        contact.type = CollisionType::Left;
    }
    else if (ballRight > WINDOW_WIDTH)
    {
        contact.type = CollisionType::Right;
    }
    else if (ballTop < 0.0f)
    {
        contact.type = CollisionType::Top;
        contact.penetration = -ballTop;
    }
    else if (ballBottom > WINDOW_HEIGHT)
    {
        contact.type = CollisionType::Bottom;
        contact.penetration = WINDOW_HEIGHT - ballB
    }

    return contact;
}
```

The penetration depth is only important when colliding with

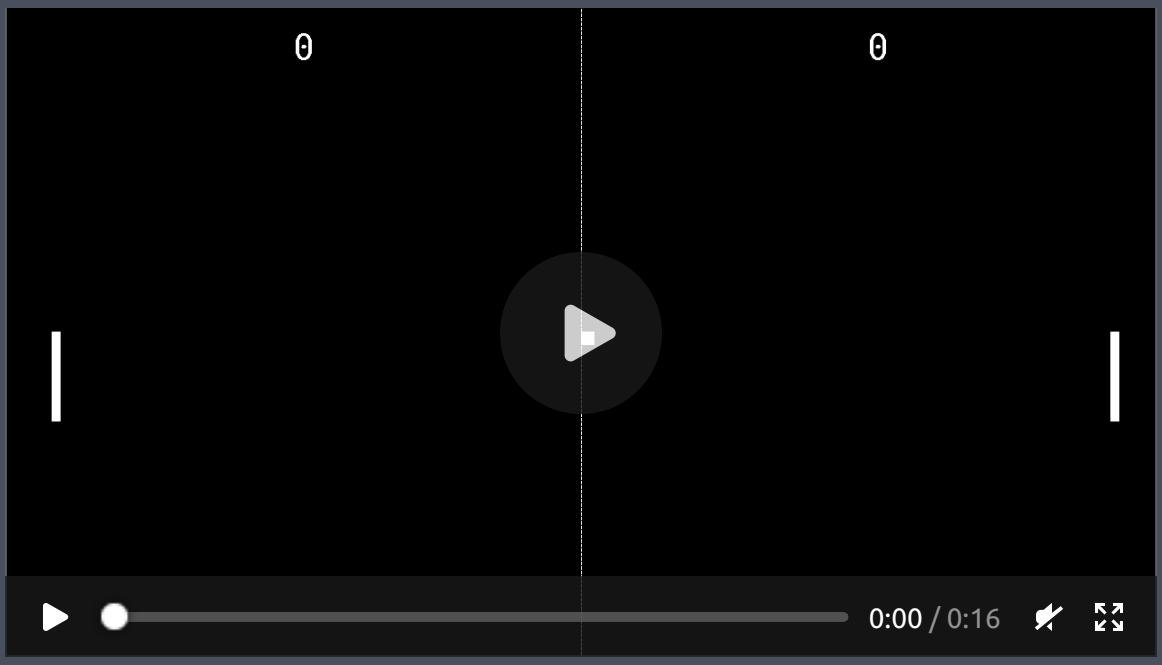
the top and bottom walls because of the bouncing, while a left or right collision just indicates that someone scored so no penetration is needed.

We'll add a method to **Ball** for wall collisions, and then check for them in the main loop.

```
[...]

class Ball
{
    [...]

    void CollideWithWall(Contact const& contact)
    {
        if ((contact.type == CollisionType::Top)
            || (contact.type == CollisionType::Bott
        {
            position.y += contact.penetration;
            velocity.y = -velocity.y;
        }
        else if (contact.type == CollisionType::Lef
        {
            position.x = WINDOW_WIDTH / 2.0f;
            position.y = WINDOW_HEIGHT / 2.0f;
            velocity.x = BALL_SPEED;
            velocity.y = 0.75f * BALL_SPEED;
        }
        else if (contact.type == CollisionType::Rig
        {
            position.x = WINDOW_WIDTH / 2.0f;
            position.y = WINDOW_HEIGHT / 2.0f;
            velocity.x = -BALL_SPEED;
            velocity.y = 0.75f * BALL_SPEED;
        }
    }

    [...]
};

[...]

// Check collisions
if (Contact contact = CheckPaddleCollision(ball, pa
    contact.type != CollisionType::None)
{
    ball.CollideWithPaddle(contact);
}
else if (contact = CheckPaddleCollision(ball, paddl
    contact.type != CollisionType::None)
{
    ball.CollideWithPaddle(contact);
```

```
46      }
47      else if (contact = CheckWallCollision(ball);
48          contact.type != CollisionType::None)
49      {
50          ball.CollideWithWall(contact);
51      }
52
53      [...]
```

If the ball goes off the left or right side of the screen, then someone has scored and the ball is reset to the middle, shooting off towards the side of the loser. If it hits the top or bottom edges, it's moved outside according to the penetration depth and is reflected.



Keeping Score

We already have the functionality that displays the scores, we now just need to actually keep track and update the score display.

Let's add a function to **PlayerScore** to allow for setting the score.

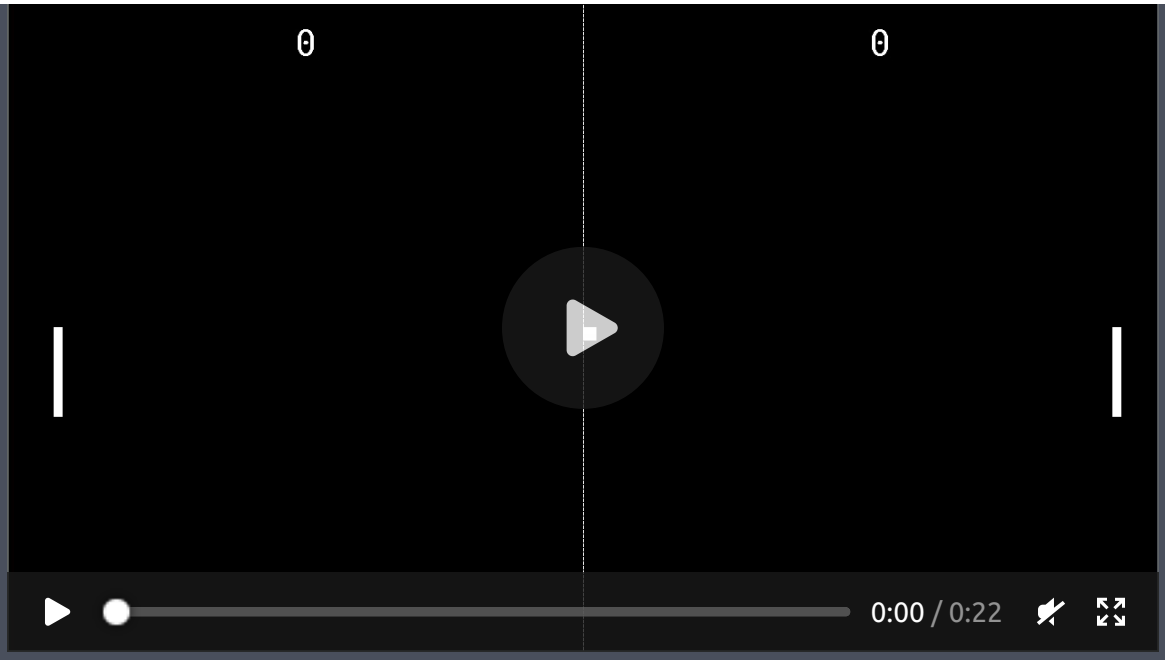
```
#include <string>

[...]

class PlayerScore
{
    [...]

    void SetScore(int score)
    {
        SDL_FreeSurface(surface);
        SDL_DestroyTexture(texture);

        surface = TTF_RenderText_Solid(font, std::t
        texture = SDL_CreateTextureFromSurface(rend

        int width, height;
        SDL_QueryTexture(texture, nullptr, nullptr,
        rect.w = width;
        rect.h = height;
    }

    [...]
};
```

It first frees the old surface and texture, and then it creates new ones from the score that is passed in.

Then we just need the logic to detect when a score occurs, increment a score counter, and display it.

```
    [...]

    int playerOneScore = 0;
    int playerTwoScore = 0;

    bool running = true;
    bool buttons[4] = {};

    float dt = 0.0f;

    [...]

    else if (contact = CheckWallCollision(ball);
        contact.type != CollisionType::None)
    {
        ball.CollideWithWall(contact);

        if (contact.type == CollisionType::Left)
        {
            ++playerTwoScore;

            playerTwoScoreText.SetScore(playerTwoScore)
        }
        else if (contact.type == CollisionType::Right)
        {
            ++playerOneScore;

            playerOneScoreText.SetScore(playerOneScore)
        }
    }
```

If the ball has a passes through the left side of the window, it's a score for Player Two. If the ball passes through the right side of the window, it's a score for Player One.



Adding Sound Effects

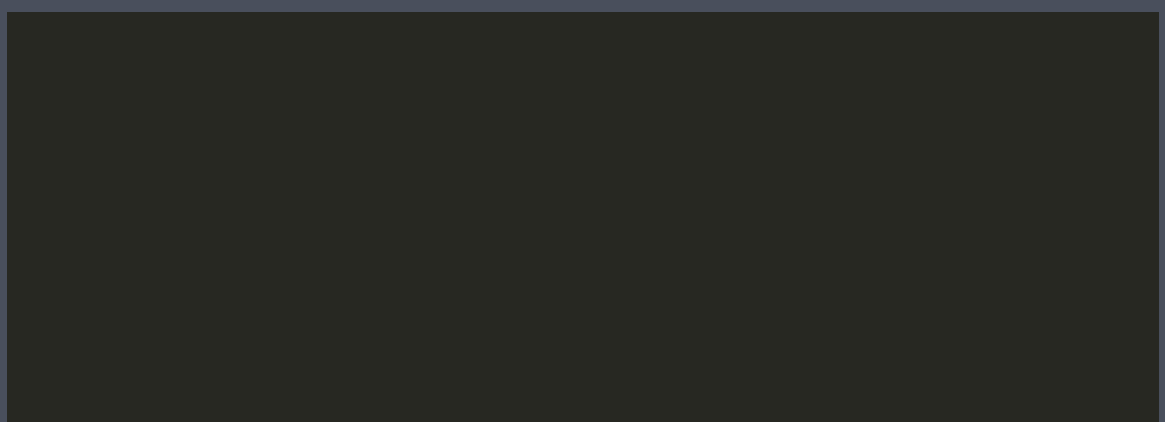
Audio is also a complicated topic, but thankfully SDL2 makes it easy enough, at least for simple sound effects. We need to initialize SDL2's audio subsystem like we did for video and text, and then create the sound effects for a paddle hit or a wall hit.

Download the two WAV files, rename them accordingly, and put them in the same directory as your executable.

- **PaddleHit.wav**
- **WallHit.wav**

Thanks to **NoiseCollector** for creating and releasing those.

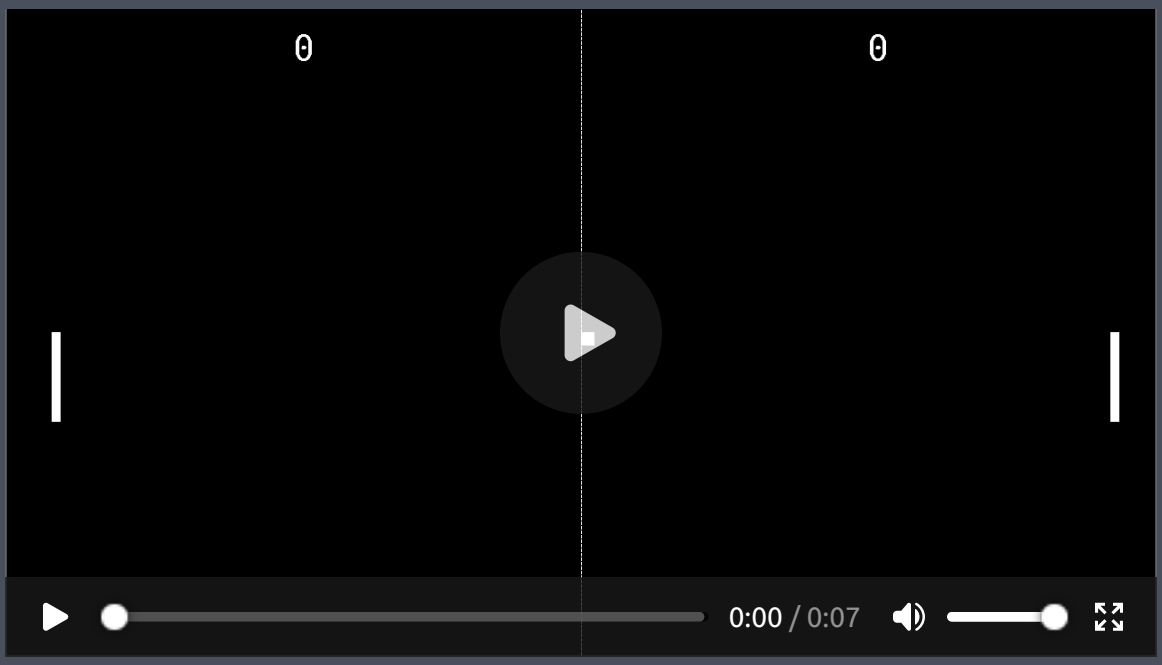
We initialize the audio subsystem with some safe defaults (see SDL2 documentation), create the sound "chunks", play them where appropriate, and free everything when we're done.



```
[...]

// Initialize SDL components
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO);
TTF_Init();
Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 2, 2048);

[...]

// Initialize sound effects
Mix_Chunk* wallHitSound = Mix_LoadWAV("WallHit.wav"
Mix_Chunk* paddleHitSound = Mix_LoadWAV("PaddleHit.

// Check collisions
if (Contact contact = CheckPaddleCollision(ball, pa
    contact.type != CollisionType::None)
{
    ball.CollideWithPaddle(contact);

    Mix_PlayChannel(-1, paddleHitSound, 0);
}
else if (contact = CheckPaddleCollision(ball, paddl
    contact.type != CollisionType::None)
{
    ball.CollideWithPaddle(contact);

    Mix_PlayChannel(-1, paddleHitSound, 0);
}
else if (contact = CheckWallCollision(ball);
    contact.type != CollisionType::None)
{
    ball.CollideWithWall(contact);

    if (contact.type == CollisionType::Left)
    {
        ++playerTwoScore;

        playerTwoScoreText.SetScore(playerTwoScore)
    }
    else if (contact.type == CollisionType::Right)
    {
        ++playerOneScore;

        playerOneScoreText.SetScore(playerOneScore)
    }
```

```
        else
        {
            Mix_PlayChannel(-1, wallHitSound, 0);
        }
    }

    [...]

    Mix_FreeChunk(wallHitSound);
    Mix_FreeChunk(paddleHitSound);
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    TTF_CloseFont(scoreFont);
    Mix_Quit();
    TTF_Quit();
    SDL_Quit();
```

It’s a simple change, but adding a little bit of sound really helps to make the game feel more alive.



Conclusion

There you have it, a simple Pong clone. Of course my goal was simplicity, so there’s a lot of room for improvement:

- Adding sprites for more interesting graphics
- Adding particle effects (e.g., ball trail)
- Adding a start menu
- Adding a delay after scoring

- Adding more interesting physics to the bounces to make them less predictable

As they say in textbooks, I'll leave those as an exercise for the reader.

Source Code

All of the code is available **here**.

I only built and ran it on Linux, but SDL2 is multiplatform, so you should be able to get it to build on anything so long as you understand your local OS's build system.

All of the code is in a single **Main.cpp** file which, in a more complicated game, would be a bad idea, but for something this simple breaking everything apart would make it harder to understand (in my humble opinion).

Last Edited: Dec 20, 2022